Análisis de Resultados-Reto4-EDA

David Burgos – 201818326

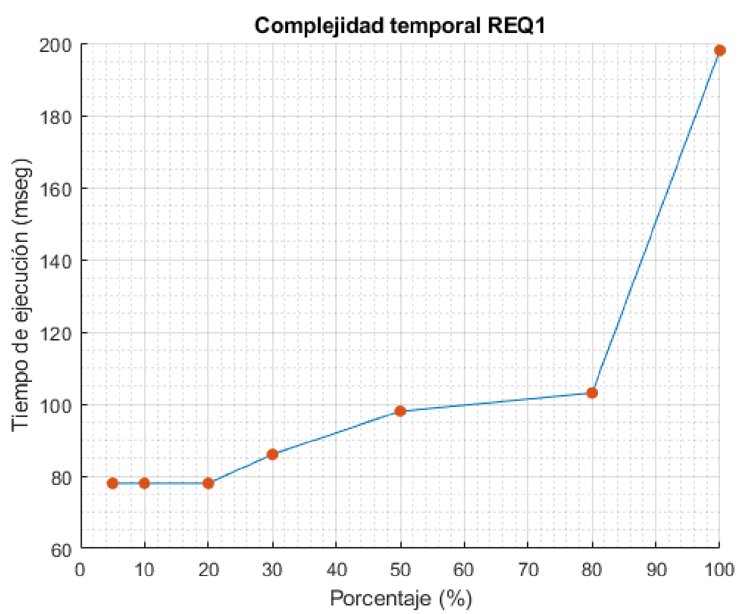
Andrés Mugnier – 201729994

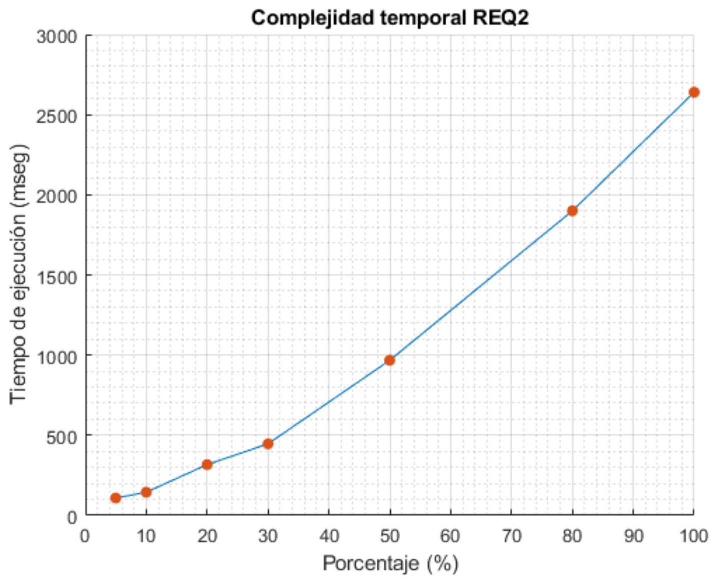
Análisis temporal experimental de los requerimientos:

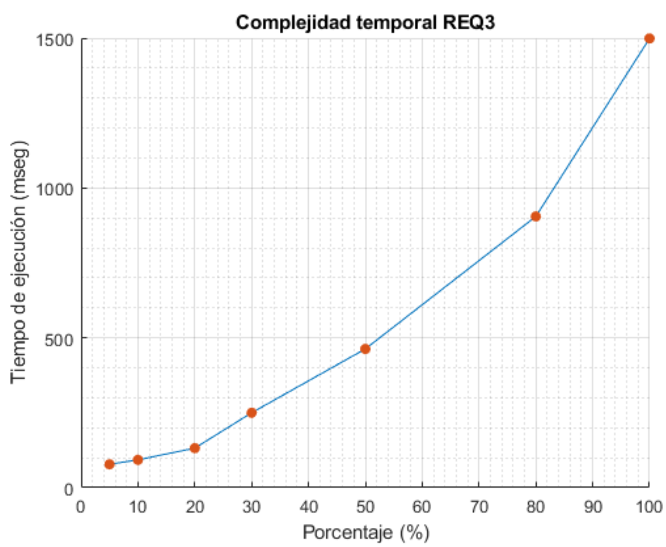
Se realizaron las pruebas de complejidad temporal para cada requerimiento utilizando los siguientes inputs:

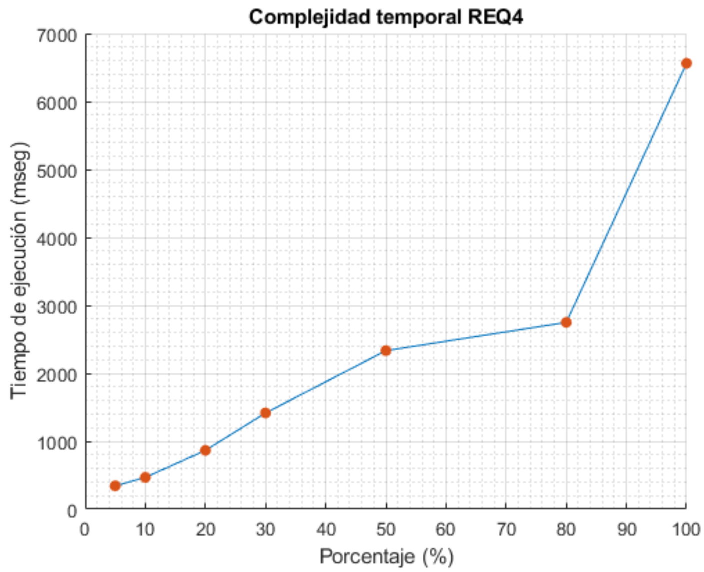
- REQ1: No hay input.
- REQ2: LED, TRP
- REQ3: St. Petesburg, Lisbon, homónimo 1.
- REQ4: LIS, 19850.0 .
- REQ5: DXB .

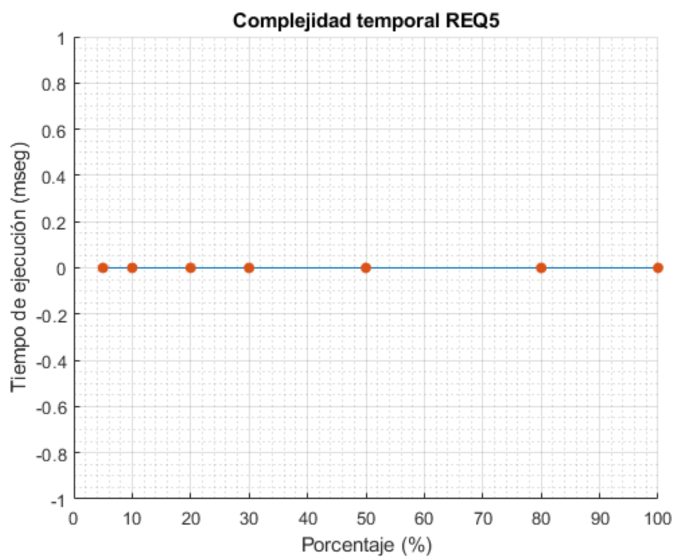
Cada prueba se realizó 3 veces para cada porcentaje de los datos propuestos. Los resultados de estas pruebas se pueden observar en las siguientes imágenes:

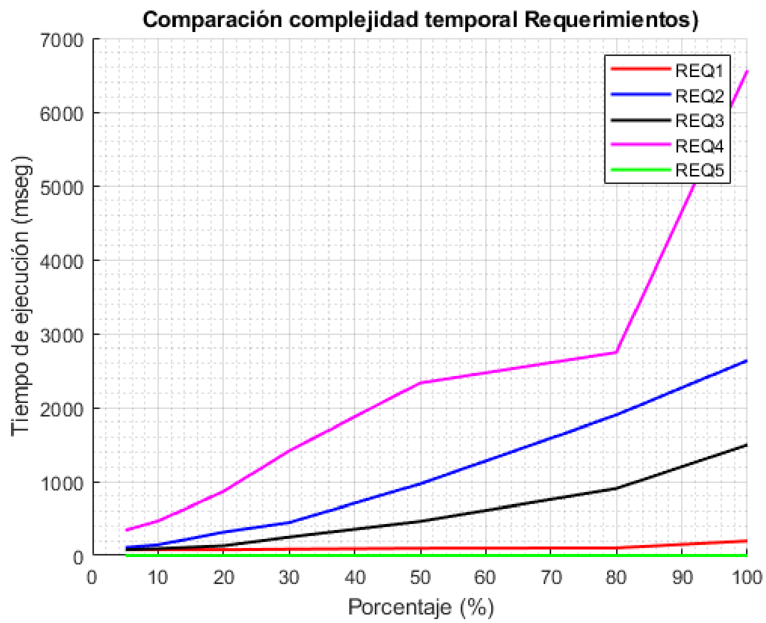



Complejidad temporal REQ2
Tiempo de ejecución (mseg)
3000
2500
2000
1500
1000
500
0
0
10
20
30
40
50
60
70
80
90
100
Porcentaje (%)


Complejidad temporal REQ3
Tiempo de ejecución (mseg)
1500
1000
500
0
0
10
20
30
40
50
60
70
80
90
100
Porcentaje (%)


Complejidad temporal REQ4
Tiempo de ejecución (mseg)
7000
6000
5000
4000
3000
2000
1000
0
0
10
20
30
40
50
60
70
80
90
100
Porcentaje (%)


Complejidad temporal REQ5
Tiempo de ejecución (mseg)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
10
20
30
40
50
60
70
80
90
100
Porcentaje (%)



Comparación complejidad temporal Requerimientos)
Tiempo de ejecución (mseg)
Porcentaje (%)
REQ1
REQ2
REQ3
REQ4
REQ5
7000
6000
5000
4000
3000
2000
1000
0
0
10
20
30
40
50
60
70
80
90
100

Análisis temporal teórico de los requerimientos:

Para todos los análisis G=(V,E)=Grafo dirigido y G’=(V’,E’)=Grado NO dirigido.

- REQ1:

```
358 def AeroInter(catalog): #REQ1
359     '''
360     Funcion que calcula el top 5 aeropuertos más interconectados
361     '''
362     #Toma el grafo dirigido
363     Graph=catalog['GRAPHD']
364     #Saca la lista de vertices
365     VertexList=gr.vertices(Graph)
366     #Crea una lista vacía para guardar el top 5                 t
367     ConnectedList=lt.newList()
368     #Contador para ir calculando el top 5
369     Top=0
370     #Recorre todos los vertices
371     for vertex in lt.iterator(VertexList):
372         #Calcula el degree de cada vertice
373         VertexDegree=gr.degree(Graph,vertex)   → O(E)
374 
375         #Va guardando y actualizando el top5
376         if VertexDegree>Top:
377             #Saca la info necesaria para imprimir en consola
378             Indegree=gr.indegree(Graph,vertex)       } O(E)
379             Outdegree=gr.outdegree(Graph,vertex)                      t·V
380             ElementInfo=mp.get(catalog['airports'],vertex)['value']
381             Name=ElementInfo['name']
382             City=ElementInfo['city']
383             Country=ElementInfo['country']
384             New={'vertex':vertex,'degree': VertexDegree,'name':Name,'city':City,'country':Country,'indegree':Indegree,'outdegree':Outdegree}
385             #Lo añade al top 5
386             lt.addLast(ConnectedList,New)
387             Top=VertexDegree
388 
389     #Sortea la lista de top 5
390     mrgsort.sort(ConnectedList,cmpByDegree)   → t
391     return ConnectedList
```

Para este REQ tenemos que todas las operaciones son constantes salvo tres. En la línea 371 hay un for que se repite V veces pues se recorren todos los vértices. Dentro de este for se calcula el degree, el indegree y el outdegree de cada vertice (Esto tiene complejidad O(E) en una estructura de datos de lista de adyacencia para cada uno). Finalmente hay un merge sort pero como la lista es constante (Siempre es el top 5) este mergesort va a ser constante.

Por lo tanto este REQ tiene una complejidad estimada de O(3*V*E) que la aproximamos a O(V*E).

- REQ2:

```
def ComponentesFuertes(catalog, ae1, ae2):
    #se saca el grafo dirigido
    principal = catalog["GRAPHD"]
    #se implementa Kosaraju y se busca si los componentes estan conectados
    kosa = scc.KosarajuSCC(principal)  O(V+E)
    cantidad = scc.connectedComponents(kosa)

    conectados = scc.stronglyConnected(kosa, ae1, ae2)

    return[cantidad, conectados]
```

En este requerimiento la operación más costosa en tiempo es el algoritmo de Kosaraju que para una lista de adyacencia corre en O(V+E).

Por lo tanto, la complejidad de estimada es O(V+E).

- REQ3:

```python
def viajeCiudades(catalog,city1,city2):

    #sacan las estructuras necesitadas
    principal = catalog["cityList"]
    Aereopuertos = catalog["airportsByCity"]
    grafo = catalog["GRAPHD"]
    #se sacan las ciudades que comparten el mismo nombre
    listaC1 = mp.get(principal, city1)["value"]
    listaC2 = mp.get(principal, city2)["value"]
```

t

```python
    C1 = None
    C2 = None

    #se determina por el usuario cual ciudad es la que se va a buscar
    if lt.size(listaC1) >1:
        print("Encontramos ciudades omonimas")
        print("")

        for x in range(lt.size(listaC1)):
            elemento = lt.getElement(listaC1, x+1)
            print(str(x+1)+ ")" + elemento["city"] + " en " + elemento["country"] + " con latitud " +  elemento["lat"] + " y longitud " +  elemento["lng"])
            print("")

        posi = int(input("Elige 1: "))
        print("")

        C1 = lt.getElement(listaC1, posi)
    else:
        C1 = lt.getElement(listaC1, 1)
```

$N_1 t$

```python
    #para ciudad 2
    if lt.size(listaC2) >1:
        print("Encontramos ciudades omonimas")
        print("")

        for x in range(lt.size(listaC2)):
            elemento = lt.getElement(listaC2, x+1)
            print(str(x+1)+ ")" + elemento["city"] + " en " + elemento["country"] + " con latitud " +  elemento["lat"] + " y longitud " +  elemento["lng"])
            print("")

        posi = int(input("Elige 1: "))
        print("")

        C2 = lt.getElement(listaC2, posi)
    else:
        C2 = lt.getElement(listaC2, 1)
```

$N_2 t$

```python
    C1Aereo = mp.get(Aereopuertos,C1["city"])["value"]
    C2Aereo = mp.get(Aereopuertos,C2["city"])["value"]

    salida = None
    llegada = None

    #se busca el aereopuerto mas cercano
    if lt.size(C1Aereo) > 1:

        val = []
        for x in range(lt.size(C1Aereo)):
            aereo = lt.getElement(C1Aereo, x+1)
            latAE = aereo["latitude"]
            longAE = aereo["longitude"]
            latC1 = C1["lat"]
            longC1 = C1["lng"]
            dist = haversine(float(longAE), float(latAE), float(longC1), float(latC1))
            val.append(dist)

        minimo = min(val)

        posi = val.index(minimo)

        salida = lt.getElement(C1Aereo, posi)
    else:
        salida = lt.getElement(C1Aereo, 1)
```

$M_1 t$

```python
    #para ciudad 2

    if lt.size(C2Aereo) > 1:

        val = []
        for x in range(lt.size(C2Aereo)):
            aereo = lt.getElement(C2Aereo, x+1)
            latAE = aereo["latitude"]
            longAE = aereo["longitude"]
            latC2 = C2["lat"]
            longC2 = C2["lng"]
            dist = haversine(float(longAE), float(latAE), float(longC2), float(latC2))
            val.append(dist)

        minimo = min(val)

        posi = val.index(minimo) + 1

        llegada = lt.getElement(C2Aereo, posi)
    else:
        llegada = lt.getElement(C2Aereo, 1)
```

$M_2 t$

```python
    #se utiliza el algoritmo Dijkstra para ecnontrar la ruta mas rapida entre los dos aereopuertos
    caminos = djk.Dijkstra(grafo, salida["IATA"])
    ruta = djk.pathTo(caminos, llegada["IATA"])

    return [ruta, salida, llegada]
```

$O(V+E)$

Si N_1 y N_2 son la cantidad de homónimos de la ciudad de salida y llegada respectivamente y M1 y M2 son la cantidad de aeropuertos de la ciudad de salida y de llegada entonces como son pequeños, podemos decir que la operación más costosa es ejecutar el algoritmo de Dijkstra que tiene complejidad O(V+E) para una lista de adyacencia.

Por lo tanto, la complejidad de este requerimiento es O(V+E).

- REQ4:

```python
216 def Miles(InputCity,Miles,catalog):
217     '''
218     Funcion que saca la rama más larga de el MST con raíz InputCity
219     '''
220     #Toma el grafo dirigido y el hashmap de la info de ciudades
221     Graph=catalog['GRAPHND']
222     CityInfoMap=catalog['airports']
223     #Calcula un MST
224     MST=prim.PrimMST(Graph)                 V'^2
225 
226     #Saca la suma de los costos de todos los arcos del MST
227     MSTCost=prim.weightMST(Graph,MST)       E_3
228     #Saca el tamaño del MST
229     NumVertex=que.size(MST['mst'])
230     #Saca la info de la ciudad de input
231     Info=mp.get(CityInfoMap,InputCity)['value']
232     #Va viendo cuál es la rama más larga
233     longest=0
234 
235     #Calcula el algoritmo de Dijkstra para la input city
236     Dijk=djk.Dijkstra(Graph,InputCity)      V'^2
237     Route=None
238 
239     #Recorre todos los nodos del MST
240     for Element in range(int(NumVertex)):
241         Element=que.dequeue(MST['mst'])
242 
243         In=Element['vertexA']
244 
245         #Calcula la mínima distancia entre la ciudad de input y el nodo del MST
246 
247                                                                             V_3
248         dist=djk.distTo(Dijk,In)
249 
250 
251         #Va actualizando la rama más larga y si sí pertenece al MST
252         if  dist > longest and dist !=float('inf') and djk.pathTo(Dijk,In) != None:
253 
254             Route=In
255             longest=dist
256 
257 
258 
259     #Calcula la ruta final
260     route=djk.pathTo(Dijk,Route)
261 
262     return MSTCost,Info,NumVertex,route
```

Para este requerimiento se utiliza el agoritmo de Prim que tiene una complejidad de V'^2 en una matriz de adyacencia (línea 224), luego se calcula el peso total del MST calculado que va a tener complejidad E_3, si E_3 es la cantidad de arcos que tiene el MST.

Luego se utiliza el algoritmo de Dijkstra que tiene complejidad V'^2 también para una matriz de adyacencia.

Finalmente se realiza un for V_3 veces recorriendo los V_3 nodos que hay en el MST.

Como un árbol de recubrimiento es muy pequeño en términos de vértices y arcos en comparación al grafo original (A menos que sea poco denso y conectado).

Así, tenemos que la complejidad temporal es O(V'^2+E_3 + V'^2 + V_3) que lo aproximamos con O(2V'^2) (por lo anterior), esto finalmente es O(V'^2).

- REQ5:

```
def AeroCerrado(catalog, cerrado):
    #se saca el grafo dirigido
    principal = catalog["GRAPHD"]
    #se sacan tando los adyacentes(Aereopuertos y vuelos afectados afectados)
    retorno1 = gr.adjacents(principal, cerrado)
    retorno2 = gr.degree(principal, cerrado)
```

Para este requerimiento se calculan los vértices adyacentes a un vertice que tiene complejidad O(E) Y luego se calcula el degree de un vértice que también tiene complejidad O(E). Por lo tanto, la complejidad de este requerimiento se aproxima como O(2E)=O(E).

- Resumen:

G=(V,E)=Grafo dirigido y G’=(V’,E’)=Grado NO dirigido.

Complejidades temporales	
Requisito	Complejidad
1	O(V*E)
2	O(V+E)
3	O(V+E).
4	O(V’^2)
5	O(E)